IERG4998IJ01
FINAL YEAR PROJECT I

PRIVACY-PRESERVING INSTANT MESSAGING ANDROID APPLICATION USING PYTHON

TSOI, MING HON 1155175123
ZHANG, YI YAO 1155174982

CONTENT

01 INTRODUCTION

02 RELATED WORK

03 METHODOLOGIES

04 EXPERIMENT PROCESS AND RESULTS

05 CONCLUSION

06 FUTURE DIRECTIONS

07 Q&A

1. INTRODUCTION

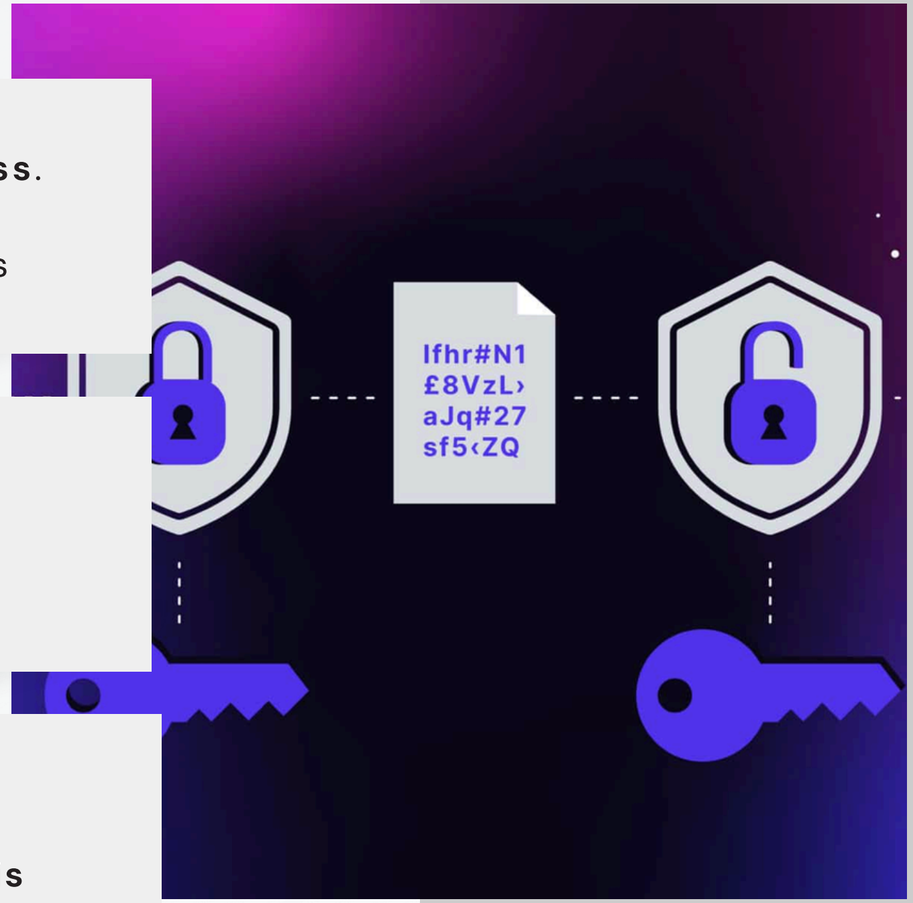
- Background
- Motivation and Significance
- Project Goals
- Project Pipeline

BACKGROUND

Messaging encryption is crucial for **protecting sensitive information from unauthorized access**. It ensures that only the intended recipients can read the messages, safeguarding against various cyber threats.

Two prominent examples of privacy-preserving instant messaging applications are **Signal** and **Apple's iMessage with PQ3**.

You share a **personal secret** with your **friend**, agreeing it’s just **between the two of you**. Suddenly, you hear **whispers**—and **your secret is out**!



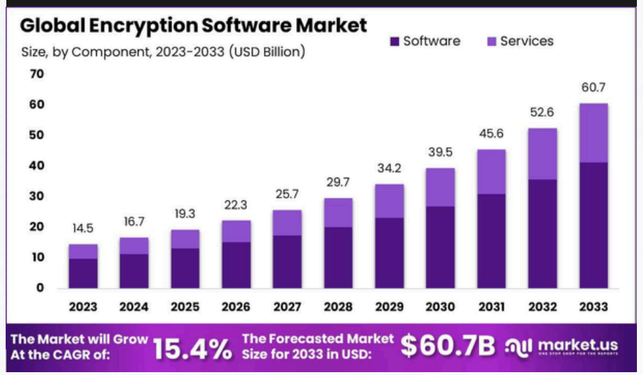
MOTIVATION AND SIGNIFICANCE

Motivation

- We **love cybersecurity** and **using tech to solve real problems**.
- We want to get **hands-on experience** with advanced ways to keep messages safe.

- **Global cyber attacks rose by 28%** in the Q3 of 2022. The global encryption software market is also expected to **grow from $5.4 billion in 2020 to $13.9 billion by 2025**, at a CAGR of 21.0%.
- The statistics are demonstrating the **rising demand** for encryption solutions.

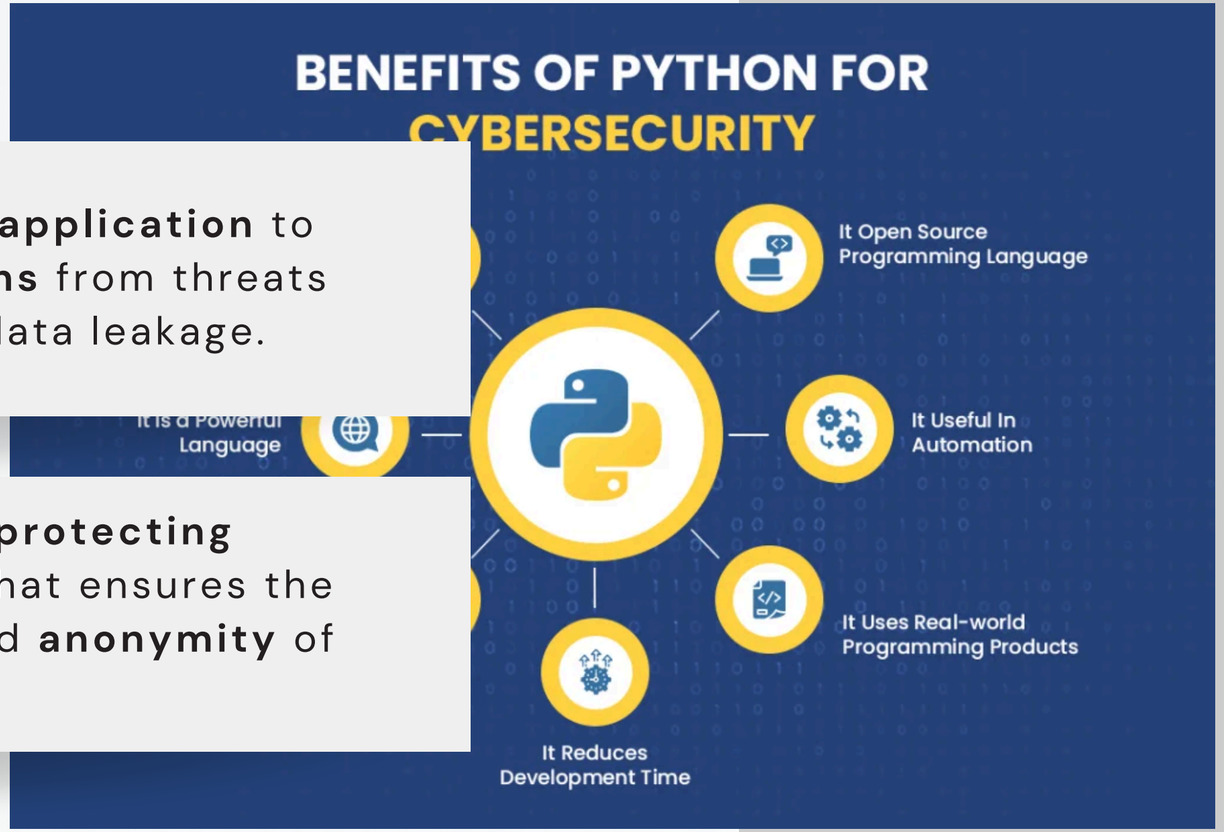
Significance



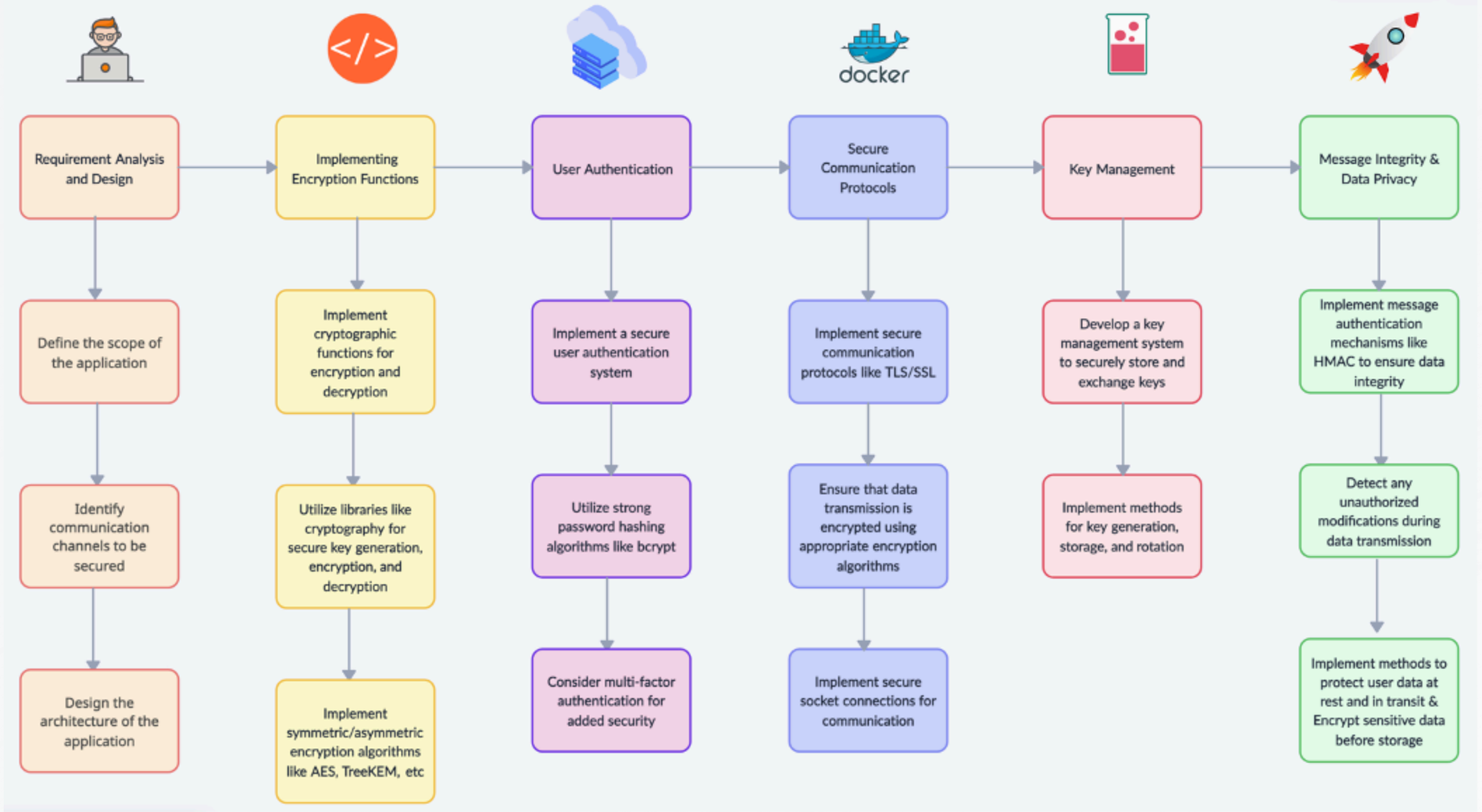
PROJECT GOALS

To develop a **Python-based application** to **protect user communications** from threats such as eavesdropping and data leakage.

We aim to create a **privacy-protecting instant messaging system** that ensures the **confidentiality**, **integrity** and **anonymity** of shared communications.



PROJECT PIPELINE



2. RELATED WORK

- Overview of Previous Research
- Analysis of Research Papers

PREVIOUS RESEARCH

Lv.1
Signal groups and **Sender Keys** offer simplicity but face scalability issues and potential vulnerabilities in ensuring post-compromise security (PCS).

Lv.2
Matrix's protocol refines the Sender Keys concept with decentralized features and improved PCS mechanisms.

Lv.3
The **MLS protocol** introduces **TreeKEM** for efficient key updates in large groups but relies on centralization and strict commit processing.

Lv.4
Re-randomized TreeKEM and **Causal TreeKEM** explore strengthening forward secrecy and handling dynamic groups in decentralized scenarios.

Ultimate Target:

To build **Concurrent TreeKEM** and **Decentralized Continuous Group Key Agreement protocol (DCGKA)**. They aim to enhance PCS update efficiency, with the latter providing advantages in post-compromise security and concurrent update handling.

Protocol	Central server not needed	Broadcast messages	Update & remove costs:[1]		PCS & FS	PCS in face of concurrent updates
			Sender	Per recipient		
Signal groups	✓[2]	✗	$\mathcal{O}(n)$	$\mathcal{O}(1)$	½ (PCS issues)[3]	[Optimal][3]
Sender Keys (WhatsApp)	✓[2]	✓	$\mathcal{O}(n)$	$\mathcal{O}(n)$	FS only[4]	[Optimal][4]
Matrix	✓	✓	$\mathcal{O}(n)$	$\mathcal{O}(n)$	PCS only	Optimal[3]
MLS (TreeKEM)	✗	✓	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	½ (FS issues)	Only one sequence heals
Re-randomized TreeKEM	✗	✓	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	✓	
Causal TreeKEM	✓[2]	✓	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	½ (severe FS issues)	Any sequence heals
Concurrent TreeKEM	✓[2]	✓	$\mathcal{O}(t + t\log(n/t))$	$\mathcal{O}(1)$	PCS only	After two rounds
Our DCGKA protocol	✓	✓	$\mathcal{O}(n)$	$\mathcal{O}(1)$	✓	All but last can be concurrent

[1] Costs are the number of public-key cryptographic operations performed. The total size of messages broadcast equals the "Sender" column except for Sender Keys and Matrix, which have total broadcast network cost $\mathcal{O}(n^2)$. t denotes the number of mutually concurrent messages.
[2] Does not specify how to determine group membership in the face of concurrent additions and removals.
[3] Optimal PCS in the face of concurrent updates is possible by using a 2-party protocol with optimal PCS+FS in place of pairwise Signal.
[4] Optimal PCS in the face of concurrent updates is possible at the given costs, but not used in practice.

Comparison of different messaging protocols

RESEARCH PAPER ANALYSIS

We have reviewed the following five research papers and all of which are closely aligned with our project.

CSIDH: AN EFFICIENT POST-QUANTUM COMMUTATIVE GROUP ACTION

The paper shows the significance of the proposed **CSIDH scheme** and provides a solution that **addresses the inefficiency of previous schemes** while maintaining security.

KEY AGREEMENT FOR DECENTRALIZED SECURE GROUP MESSAGING WITH STRONG SECURITY GUARANTEES.

The paper introduces a **Decentralized Continuous Group Key Agreement (DCGKA)** security notion for establishing **shared symmetric keys in dynamic groups**.

PARAKEET: PRACTICAL KEY TRANSPARENCY FOR END-TO-END ENCRYPTED MESSAGING

The paper introduces **Parakeet**, a **key transparency system** designed to **address scalability issues** that limit previous academic works from practical implementation.

SECURITY ANALYSIS OF THE MLS KEY DISTRIBUTION

The paper discusses the **challenges of group communication protocols**, the importance of continuous key- exchange for dynamic groups, and the specific focus on the **MLS continuous group key distribution (CGKD)**, including the **MLS key schedule and TreeKEM** design.

HOW TO WATERMARK CRYPTOGRAPHIC FUNCTIONS

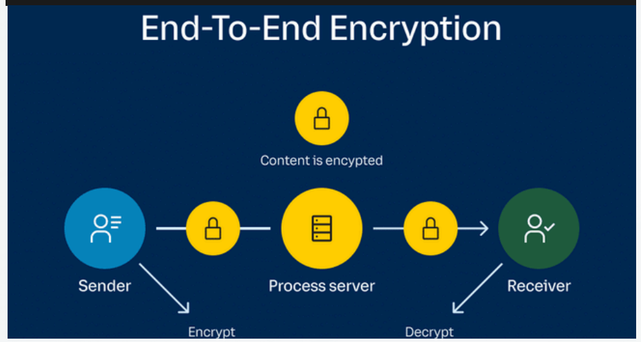
This paper discusses a way to **add hidden marks to special math rules** that keep secrets safe. These marks help **identify copies and protect the rules** from being changed by bad actors.

3. METHODOLOGIES

- End-to-End Encryption
- E2EE Methods

WHAT IS END-TO-END ENCRYPTION (E2EE)?

E2EE?



- E2EE is a private communication system that **prevents data from being read or secretly modified** by people **other than the true sender and recipient**.
- In E2EE, messages are **encrypted by the sender**, but **third parties cannot decrypt** them and store them in an encrypted manner.

- **Signal** and **Apple's iMessage** are two widely used messaging apps that use end-to-end encryption.
- **Signal** uses the **Signal protocol**, which combines **Diffie-Hellman key exchange**, **symmetric key encryption**.
- **Apple's iMessage** recently launched the **PQ3 encryption protocol**, which combines **classical elliptic curve encryption** with **post-quantum Kyber encryption**.

Examples

E2EE METHODS

01

DIFFIE-HELLMAN KEY EXCHANGE

A **shared secret key** is established between communicating parties

02

AES

A **symmetric-key algorithm** that is used by WhatsApp and Signal. The algorithm processes **data in blocks**, with each block undergoing a **specific number of rounds** depending on the **key size**.

03

RSA

An **asymmetric-key algorithm** which the implementation of it makes heavy use of **modular arithmetic**, **Euler's theorem**, and **Euler's totient function**.

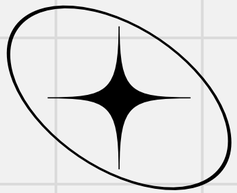
04

SIGNAL PROTOCOL

An **open-source encryption protocol** developed by **Open Whisper Systems**. It combines several cryptographic primitives, including the **Double Ratchet algorithm**.

4. EXPERIMENT PROCESS AND RESULTS ANALYSIS

1. TREENODE CLASS & MLS (KEY MANAGEMENT, ADDING/REMOVING MEMBERS, KEY GENERATION)
2. AES ENCRYPTION AND CURVE25519 KEYS
3. TEXT AND PHOTO ENCRYPTION
4. WATERMARKING
5. SAVING ENCRYPTED DATA
6. LOADING THE DERIVED KEY
7. AES DECRYPTION
8. TEXT AND PHOTO DECRYPTION
9. WATERMARK EXTRACTION



DEMONSTRATION TIME

TIME FOR ENCRYPTION!

```python
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import padding
import shlex
import os

# TreeKEM
class TreeNode:
    def __init__(self):
        self.children = []
        self.group_key = None

    def add_member_TreeNode(self, new_public_key):
        new_node = TreeNode()
        new_node.public_key = new_public_key
        self.children.append(new_node)
        self.update_key_TreeNode()

    def remove_member_TreeNode(self, public_key_to_remove):
        for child in self.children:
            if child.public_key == public_key_to_remove:
                self.children.remove(child)
                self.update_key_TreeNode()
                break

    def update_key_TreeNode(self):
        self.group_key = self.generate_group_key_TreeNode()

    def generate_group_key_TreeNode(self):
        if not self.children:
            return os.urandom(32)

        combined_key = b''.join(child.generate_group_key_TreeNode() for child in self.children)
        return HKDF(
            algorithm=hashes.SHA256(),
            length=32,
            salt=os.urandom(16),
            info=b'',
        ).derive(combined_key)

    def print_tree(self, level=0):
        if not self.children:
            print("  " * level + f"Leaf Node Level {level}: Public Key: {self.public_key.public_
        else:
            print("  " * level + f"Node Level {level}: Group Key: {self.group_key.hex()}")

        for child in self.children:
            child.print_tree(level + 1)
```

TreeNode: Represents a **node in a tree structure** where **group keys** are updated and maintained.

Adds a member with a **new public key**.

Removes a member based on their **public key**.

Generates a group key based on the **children's keys**.

Prints the tree structure with **group keys and public keys**.

TREENODE CLASS & MLS

Key Management: **Each node** in the tree represents a **group member** and holds a **group key** that is **derived from its children's keys**. This hierarchical approach allows for efficient key updates.

Adding/Removing Members: The functions add_member_TreeNode and remove_member_TreeNode dynamically **update the group key** when members join or leave.

Key Generation: The generate_group_key_TreeNode method **creates a new group key using HKDF** (HMAC-based Key Derivation Function), ensuring that even when **members change**, the **group key remains secure and unique**.

```python
action = input("Would you like to add or remove members? (add/remove/no): ")
if action == "add":
    new_private_key = x25519.X25519PrivateKey.generate()
    new_public_key = new_private_key.public_key()
    root.add_member_TreeNode(new_public_key)
    print("New member added.")

    root.update_key_TreeNode()

    # Re-encrypt the photo with watermark if applicable
    if choice == 'photo' and watermark_info:
        sender_id, receiver_id = watermark_info
        encrypt_and_save_photo(root, photo_path, sender_id, receiver_id, add_watermark=True)
    elif choice == 'text':
        encrypt_and_save_text(root, message)

    root.print_tree()

elif action == "remove":
    index_to_remove = int(input("Enter the index of the member to remove (starting from 0): "))
    if 0 <= index_to_remove < len(root.children):
        root.remove_member_TreeNode(root.children[index_to_remove].public_key)
        print("Member removed.")

        root.update_key_TreeNode()

        # Re-encrypt the photo with watermark if applicable
        if choice == 'photo' and watermark_info:
            sender_id, receiver_id = watermark_info
            encrypt_and_save_photo(root, photo_path, sender_id, receiver_id, add_watermark=True)
        elif choice == 'text':
            encrypt_and_save_text(root, message)

        root.print_tree()
    else:
        print("Invalid index.")

elif action == "no":
    print("No changes were made to the members.")
else:
    print("Invalid input. Please enter 'add', 'remove', or 'no'.")
```

The user chooses to **add members**.

The user chooses to **remove members**.

TREENODE CLASS & MLS

Adding Members:
A **new X25519 private key** is **generated** using x25519.X25519PrivateKey.generate() and its **corresponding public key is extracted**.
The **new public key is added as a member to the tree** structure represented by the root object.
The update_key_TreeNode() method is **called on the root to update the keys** in the tree structure.
Depending on the user's initial choice (choice), if it's a photo and a **watermark is present**, the photo is **re-encrypted**. If it's a **text message**, the text is **re-encrypted**.
The tree structure is printed after the operation.

Removing Members:
The user is prompted to **enter the index of the member they want to remove**.
If the index is valid (within the range of existing members), the **selected member is removed from the tree structure**.
The **keys** in the tree structure are **updated** after the removal.
Similar to adding members, if applicable, the **photo is re-encrypted with the watermark or the text is re-encrypted**.
The tree structure is **printed** after the operation.

AES ENCRYPTION AND CURVE25519 KEYS

AES: It uses AES (Advanced Encryption Standard) **with CBC** (Cipher Block Chaining) mode for **symmetric encryption**.

Padding: Padding is **applied before encryption** to handle data of varying lengths.

Curve25519: It is an **elliptic curve** used in **elliptic-curve cryptography (ECC)** offering 128 bits of security (256-bit key size) and designed for use with the **Elliptic-curve Diffie–Hellman (ECDH) key agreement scheme**. And the curve used is $y^2 = x^3 + 486662x^2 + x$. It is one of the **fastest curves in ECC**, and is **not covered by any known patents**.

Encrypts data using **AES encryption** with a **given key**.

```python
# AES Functions
def aes_encrypt(data, key):
    iv = os.urandom(16)  # Initialization vector
    cipher = Cipher(algorithms.AES(key), modes.CBC(iv), backend=default_backend())
    encryptor = cipher.encryptor()

    padder = padding.PKCS7(algorithms.AES.block_size).padder()
    padded_data = padder.update(data) + padder.finalize()

    encrypted = encryptor.update(padded_data) + encryptor.finalize()
    return iv + encrypted  # Prepend IV to the encrypted data
```

Saves a **derived key** to a file.

```python
def save_derived_key(derived_key, filename):
    with open(filename, "wb") as file:
        file.write(derived_key)
```

```python
# Main
if __name__ == "__main__":
    root = TreeNode()

    choice = input("What would you like to encrypt? Enter 'text' or 'photo': ").strip().lower()
    num_members = int(input("Number of members in the group: "))

    for _ in range(num_members):
        new_private_key = x25519.X25519PrivateKey.generate()
        new_public_key = new_private_key.public_key()
        root.add_member_TreeNode(new_public_key)

    root.update_key_TreeNode()
```

Add the public key to a tree structure represented by the **root node**.

Generating a **new key pair** for **each member** using the **X25519 algorithm**

```python
def encrypt_long_message(message, derived_key):
    encrypted_chunks = []
    message_bytes = message.encode()

    encrypted_message = aes_encrypt(message_bytes, derived_key)
    encrypted_chunks.append(encrypted_message)

    return encrypted_chunks

def encrypt_photo(photo_path, derived_key):
    with open(photo_path, "rb") as photo_file:
        photo_data = photo_file.read()

    encrypted_chunks = []

    encrypted_photo = aes_encrypt(photo_data, derived_key)
    encrypted_chunks.append(encrypted_photo)

    return encrypted_chunks
```

Encrypts a **long message** using a **derived key**.

Encrypts a **photo** file using a **derived key**.

```python
if choice == 'text':
    message = input("Please enter the text message you want to encrypt: ")
    encrypt_and_save_text(root, message)
elif choice == 'photo':
    photo_path_input = input("Please enter the path of the photo you want to encrypt: ")
    photo_path = shlex.split(photo_path_input)[0]
```

It allows the user to **choose between encrypting a text message or a photo**. Depending on the user's choice, the program **prompts the user for input.**

TEXT AND PHOTO ENCRYPTION

Text Encryption flow:

It **takes a message** and a **derived key** as input and **encodes the message into bytes**.
It **encrypts** the message using the **AES encryption algorithm** with the provided **derived key**.
And the **encrypted message** is then **appended to a list encrypted_chunks** and returned.

Photo Encryption flow:

It **takes the path to a photo file** and a **derived key** as input.
It **reads** the **binary data** from the photo file.
It then **encrypts** the photo data using **AES encryption** with the provided **derived key**.
And the **encrypted photo data** is added to a **list encrypted_chunks** and returned.

WATERMARKING

Watermark:
It **reads a photo file**, **embeds a watermark** containing sender and receiver identifiers, **encrypts** the **photo data** with the **watermark** using **AES encryption**, and returns the **encrypted chunks**.

Main Function:
It allows the user to **choose whether to add a watermark** to an encrypted photo, **prompts for sender and receiver** identities if a **watermark is requested**, **encrypts and saves the photo** accordingly, **retains watermark information** if provided, and then **prints** the tree structure held in the **root object**.

```python
def encrypt_photo_with_watermark(photo_path, derived_key, sender_id, receiver_id):
    with open(photo_path, "rb") as photo_file:
        photo_data = photo_file.read()

    watermark = f"{sender_id}:{receiver_id}".encode()
    photo_data_with_watermark = photo_data + b"START_WATERMARK::" + watermark + b"::END_WATERMARK"

    encrypted_chunks = []

    encrypted_photo = aes_encrypt(photo_data_with_watermark, derived_key)
    encrypted_chunks.append(encrypted_photo)

    return encrypted_chunks
```

Encrypt a photo while **embedding a watermark** that **contains sender and receiver identifiers**.

```python
    add_watermark = input("Would you like to add a watermark? (yes/no): ").strip().lower()

    if add_watermark == 'yes':
        sender_id = input("Please enter the sender's identity: ")
        receiver_id = input("Please enter the receiver's identity: ")
        encrypt_and_save_photo(root, photo_path, sender_id, receiver_id, add_watermark=True)
        watermark_info = (sender_id, receiver_id)  # Save watermark info for later use
    else:
        encrypt_and_save_photo(root, photo_path, add_watermark=False)
        watermark_info = None  # No watermark to retain
else:
    print("Invalid input. Please enter 'text' or 'photo'.")
    exit()

root.print_tree()
```

It **prompts** the user to decide **whether to add a watermark** to an encrypted photo.

Encrypts and **saves** a **text message** using the **group key** from a **given root node**.

Encrypts and **saves** a **photo file** using the **group key** from a **given root node**.

```python
def encrypt_and_save_text(root, message):
    derived_key = root.group_key

    encrypted_chunks = encrypt_long_message(message, derived_key)

    save_derived_key(derived_key, "derived_key.bin")

    with open("encrypted_messages.txt", "wb") as file:
        for encrypted_chunk in encrypted_chunks:
            file.write(encrypted_chunk)

    print("Encrypted Message chunks:")
    for i, encrypted_chunk in enumerate(encrypted_chunks):
        print(f"Chunk {i + 1}: {encrypted_chunk}")

def encrypt_and_save_photo(root, photo_path, sender_id=None, receiver_id=None, add_watermark=False):
    derived_key = root.group_key

    if add_watermark:
        encrypted_chunks = encrypt_photo_with_watermark(photo_path, derived_key, sender_id, receiver_id)
    else:
        encrypted_chunks = encrypt_photo(photo_path, derived_key)

    save_derived_key(derived_key, "derived_key.bin")

    with open("encrypted_photo.bin", "wb") as file:
        for encrypted_chunk in encrypted_chunks:
            file.write(encrypted_chunk)

    print("The encrypted photo has been saved to 'encrypted_photo.bin'.")
```

SAVING ENCRYPTED DATA

Text:
It **retrieves** the **derived key** from the **root object**.
It **saves the derived key** to a file named "**derived_key.bin**".
It then **iterates over the encrypted chunks**, **writes** them to a file named "**encrypted_messages.txt**", and prints each chunk.

Photo:
It **obtains** the **derived key** from the **root object**.
The **derived key** is **saved** to a file named "**derived_key.bin**".
The **encrypted photo chunks** are written to a file named "**encrypted_photo.bin**".
A **message is printed** indicating that the encrypted photo has been saved.

ANOTHER DEMONSTRATION

TIME TO DECRYPT!!!

LOADING THE DERIVED KEY

```python
def load_derived_key(filename):
    with open(filename, "rb") as file:
        derived_key = file.read()
        return derived_key
```

It reads a **binary file** and returns its contents as a **binary key**.

AES DECRYPTION

It **decrypts** data encrypted with **AES (Advanced Encryption Standard) algorithm in CBC (Cipher Block Chaining) mode**, using the **provided key** and **IV**, and **removes any padding** to obtain the original **plaintext** data.

```python
def aes_decrypt(encrypted_data, key):
    iv = encrypted_data[:16]
    cipher = Cipher(algorithms.AES(key), modes.CBC(iv), backend=default_backend())
    decryptor = cipher.decryptor()

    decrypted_padded = decryptor.update(encrypted_data[16:]) + decryptor.finalize()

    unpadder = padding.PKCS7(algorithms.AES.block_size).unpadder()
    decrypted_data = unpadder.update(decrypted_padded) + unpadder.finalize()

    return decrypted_data
```

It **decrypts data** encrypted with AES in CBC mode, using the **provided key and IV**

TEXT AND PHOTO DECRYPTION

```python
def decrypt_message(encrypted_data, derived_key):
    decrypted_chunk = aes_decrypt(encrypted_data, derived_key)
    return decrypted_chunk.decode('latin-1')

def decrypt_photo(encrypted_data, derived_key):
    return aes_decrypt(encrypted_data, derived_key)

def decrypt_and_save_text():
    derived_key = load_derived_key("derived_key.bin")
    with open("encrypted_messages.txt", "rb") as file:
        encrypted_data = file.read()  # Read all data at once
    decrypted_message = decrypt_message(encrypted_data, derived_key)
    print("Decrypted message:", decrypted_message)

def decrypt_and_save_photo():
    derived_key = load_derived_key("derived_key.bin")
    with open("encrypted_photo.bin", "rb") as file:
        encrypted_data = file.read()  # Read all data at once
    decrypted_photo = decrypt_photo(encrypted_data, derived_key)

    with open("decrypted_photo.png", "wb") as file:
        file.write(decrypted_photo)
    print("The photo has been decrypted and saved as 'decrypted_photo.png'.")

    return "decrypted_photo.png"
if __name__ == "__main__":
    choice = input("What would you like to decrypt? Enter 'text' or 'photo': ").strip().lower()

    if choice == 'text':
        decrypt_and_save_text()
    elif choice == 'photo':
        decrypted_photo_path = decrypt_and_save_photo()
        watermark_info = extract_watermark(decrypted_photo_path)
        if watermark_info:
            print("Extracted watermark information:", watermark_info)
    else:
        print("Invalid input. Please enter 'text' or 'photo'.")
```

It **decrypts** the encrypted_data using the **derived_key** by **calling the aes_decrypt function** and then decodes the decrypted data using the **Latin-1 encoding** before returning it.

It **decrypts** the encrypted_data by **calling the aes_decrypt function** and returns the **decrypted binary data** without any decoding.

It **loads** the **derived key** from a file then **reads encrypted message data** from a file, decrypts it using **decrypt_message**, **decodes** it, and **prints** the decrypted message to the console.

It **loads** the **derived key** then **reads encrypted photo data**, **decrypts** it, and saves the **decrypted photo data** to a new file named "decrypted_photo.png" in **binary mode**.

It defines **user behaviours**, and choosing between decrypting **a text message or a photo**, executing the **corresponding decryption functions**.

WATERMARK EXTRACTION

The function **reads a binary file** containing a photo, searches for **specific markers** to **identify the watermark**, **extracts** the watermark content, **decodes** it as a **UTF-8 string**, and **returns** the extracted watermark information.

```python
def extract_watermark(photo_path):
    with open(photo_path, "rb") as file:
        photo_data = file.read()

    start_separator = b"START_WATERMARK::"
    end_separator = b"::END_WATERMARK"

    start_index = photo_data.find(start_separator)
    end_index = photo_data.find(end_separator, start_index)

    if start_index == -1 or end_index == -1:
        print("Watermark not found.")
        return None

    watermark_bytes = photo_data[start_index + len(start_separator):end_index]
    watermark = watermark_bytes.decode('utf-8', errors='ignore')

    return watermark
```

It **extracts watermark information** from a **binary file** located at the photo_path.

5. CONCLUSION

CONCLUSION

In this semester, my project partner and I have focused on **privacy-enhancing cryptography** for **secure messaging**, and we have **reviewed several research papers** for insights.

CONCLUSION N°1

We **developed a plan** for a "Privacy-Preserving Instant Messaging" project, **explored encryption methods**, and **finalized our proposal**.

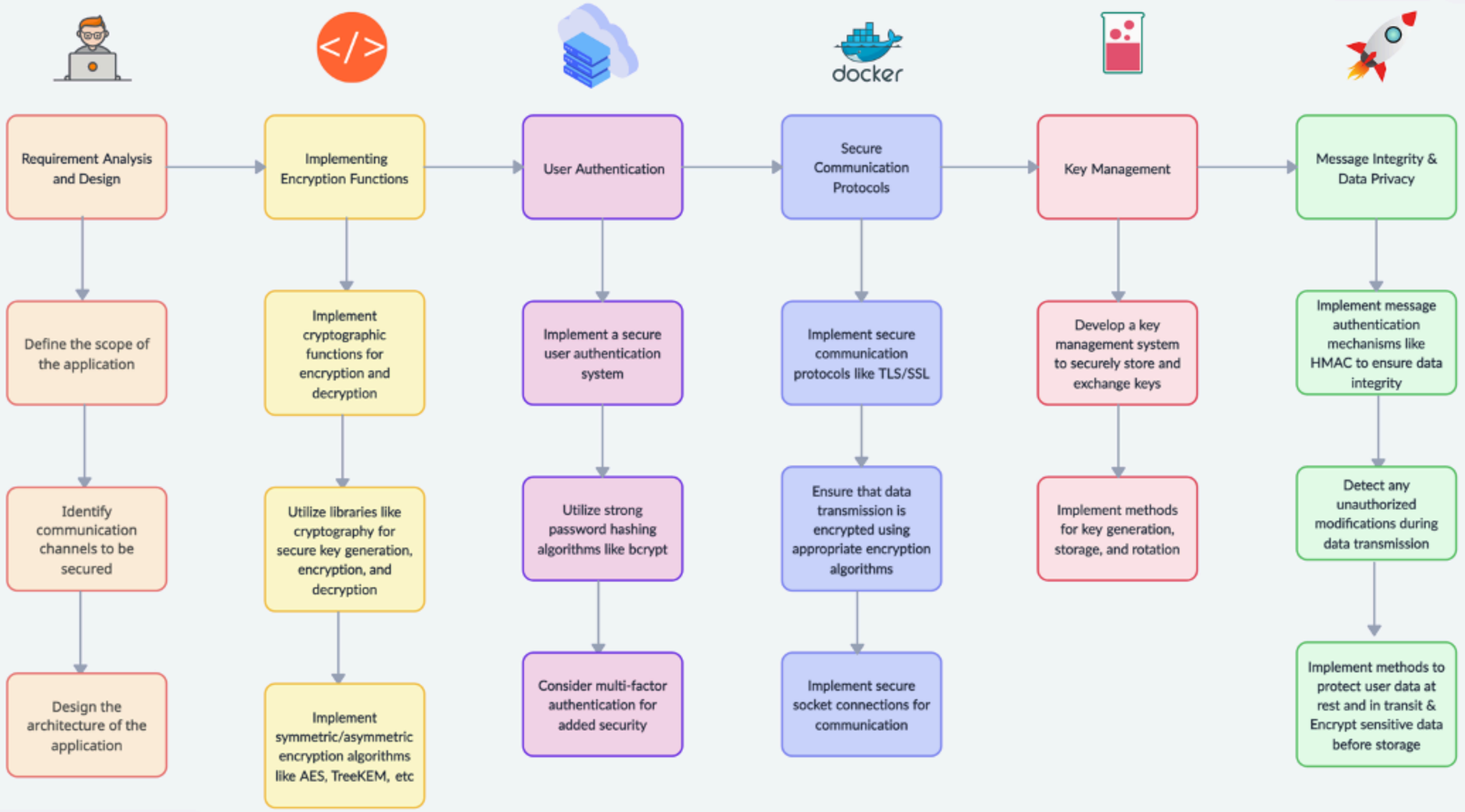
CONCLUSION N°2

We have **successfully implemented** the **Watermark encryption** and **MLS Key Distribution system** to fortify the security of our messages.

CONCLUSION N°3

6. FUTURE DIRECTIONS

FUTURE DIRECTIONS



We will continue to **improve our MLS Key Distribution System** as it is **not in its final state yet**.

At this moment, our **MLS Key distribution algorithm** and **watermark** have been **successfully integrated**. Our focus will now shift towards **building our messaging app**, refining the app's **security features**, and **optimizing user experience**. We will be **incorporating the system into the messaging app** and extensively **testing its functionality** to ensure **secure message exchanges** between users.

Q&A